FC.

8

ADA025173

Copying List Structures Without Auxiliary Storage

Douglas W. Clark

October 1975

See 1473

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Copying List Structures Without Auxiliary Storage

Douglas W. Clark

October 1975

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

This work was done while the author was supported by a grant from the Xerox Corporation Palo Alto Research Center. Partial support was provided by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research.

ABSTRACT

An algorithm is presented for copying an arbitrary list structure into a block of contiguous storage locations without destroying the original list. Apart from a fixed number of program variables, no auxiliary storage, such as a stack, is used. The algorithm needs no mark bits and operates in linear time. It is shown to be significantly faster than the best previous algorithm for the same problem.

1. Introduction

The problem considered in this paper is the creation of a copy of an arbitrary LISP-type list structure without the use of a stack or any other working storage which depends on the size or complexity of the list to be copied. Apart from a fixed number of program variables, the only storage available is that occupied by the original list and the copy. Copying differs from *moving* a list [1] in that the original structure may not be destroyed during processing.

Algorithms for the constant workspace copying problem have been given by Lindstrom [7] and Fisher [4]. Lindstrom showed how to copy an arbitrary n-cell list structure in time $O(n \log n)$ if a mark bit is available in each cell, and in time $O(n^2)$ if there are no mark bits; both algorithms can copy into an arbitrary list of available cells. Fisher's algorithm takes only linear time and needs no mark bits, but makes a minor sacrifice in generality: the free list must be a block of contiguous cells.

The algorithm to be presented in this paper is significantly faster than Fisher's, and has the same free-list restriction. While Fisher's algorithm requires three passes over the data, the algorithm of this paper requires slightly more than two full passes, depending on the degree of sharing in the structure to be copied. In addition, a pass here is more efficient than in Fisher's algorithm, especially when there are many pointers to atoms.

The principal difficulty in copying lists is that several pointers can point to the same cell. The algorithm will therefore be introduced in three stages which reflect the complexity of these multiple references. In the following section an algorithm is given for copying lists when there is no sharing at all. (This algorithm is also presented in [3]. It appears here for expository purposes.) Section 3 extends this algorithm to cover an intermediate case, and in Section 4 arbitrary list structures are allowed. A comparison of the algorithm of Section 4 with Fisher's algorithm is offered in Section 5 and the Appendix. Section 6 concludes the paper.

2. Copying a binary tree

Following LISP conventions [8], assume that a *list cell* contains two pointers, called *car* and *cdr*, which may point to any list cell or to non-list items called *atoms*. Atoms themselves are not copied. Each list cell occupies one memory location. In this paper all algorithms trace lists in the *cdr direction*; that is, they trace cdr before car if both point to lists.

We will assume in this section that the list structure to be copied contains no multiply-referenced cells, and is therefore a binary tree with non-link information at the leaf nodes only. This assumption greatly simplifies the copying task. Observe that cells with at least one atom can be dealt with in a single visit: atoms are copied directly, and the list pointer (if there is one) in the copy will point to the next consecutive cell in the free area.

Clearly, then, the only difficulty in copying a structure of this kind is what to do when a cell contains two list pointers. Were sufficient auxiliary storage available, such cells could be stacked [5], but this would violate the constant workspace constraint.

We can evade this problem by keeping a stack in the copy of the structure itself. Figure 1 shows how this can be done. The list structure to be copied is shown in Figure 1(a). In Figure 1(b) cdr-direction tracing has stopped temporarily at a cell with two atoms. The first four cells of the list have been visited, and copies of cells with at least one atom have their final values. The variable *avail* points to the next free cell in the copy area.

Copies of cells with two list pointers have been linked together in LIFO order on the "k-list": this is a linked stack of cells on which more work needs to be done. The cars of such cells have been copied without change; this permits tracing of the lists they point to when the k-list is "popped". As further lists are traced, the k-list will grow and shrink exactly like a stack. Whenever tracing stops at a cell with two atoms, car of the first cell on the k-list is the next sublist to be traced.

Two crucial observations need to be made about cells on the k-list. First, the final value of cdr of these cells is a pointer to the next sequential cell in memory. This redundancy is what permits the k-list links to be kept in the new cdrs. Second, the final value of car of such a cell is the value of avail at the time the cell is removed from the k-list. This is apparent in Figure 1(b). Therefore both car and cdr may be given their final values when the cell is popped.

Figure 1(c) shows what happens after the first k-list cell is removed and the corresponding sublist traced. Figure 1(d) shows the structure after the last k-list cell has been removed and copying has finished.

Algorithm I, below, copies the tree structure pointed to by h into the block of free storage beginning at location avail. On termination of the algorithm, v will point to

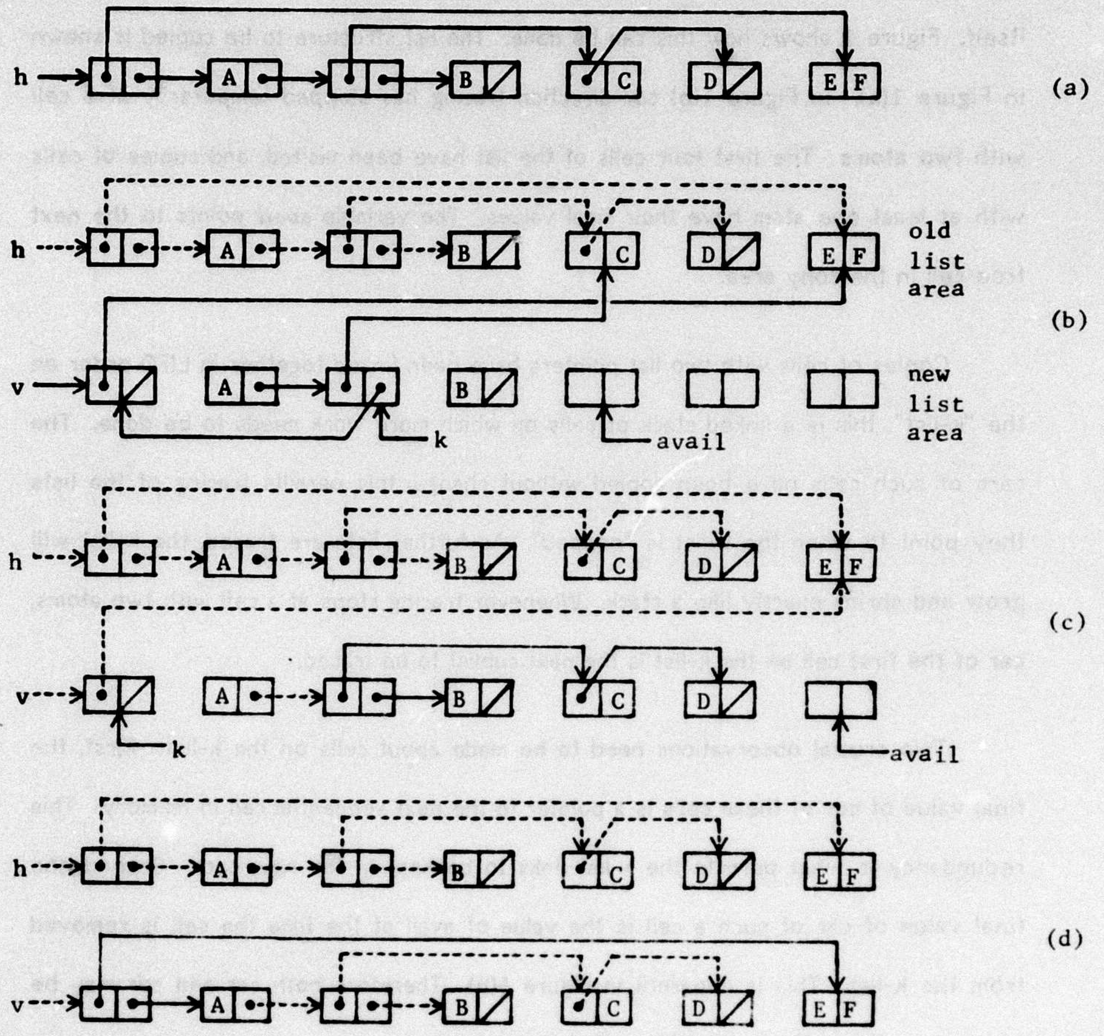


Figure 1: Copying a binary tree. Cars are on the left, cdrs are on the right, and NIL is represented by a diagonal slash. Dotted pointers are those unchanged from the previous figure.

the copy, and avail to the next available cell in the copy area. Algorithm I is significantly faster than the one given by Lindstrom [6], which requires three visits to each cell; Lindstrom's algorithm, however, can copy into an arbitrary free-list.

ALGORITHM I

Part A: Copy the top level of a list.

A1. [Initialize.] k←NIL, v←avail, and x←h. (x points to the current original cell.)

A2. [Get new cell.] n←avail, avail←avail+1. (n points to the copy of x.)

A3. [Process cell x.] Find the data types of car(x) and cdr(x). Go to the appropriate box below:

		cdr(x)	
		atom	list

car(x)	atom	car(n)←car(x)
		cdr(n)←cdr(x)
		go to B1
			car(n)←car(x)
			cdr(n)←avail
			x←cdr(x)

	list	car(n)←avail
		cdr(n)←cdr(x)
		x←car(x)
			car(n)←car(x)
			cdr(n)←k
			k←n
			x←cdr(x)

A4. [Loop.] Go to Step A2.

Part B: Find the most recently seen sublist.

B1. [Done?] If k=NIL then halt.

B2. [Pop k-list.] t←cdr(k), x←car(k), car(k)←avail, cdr(k)←k+1, k←t, and go to A2.

3. Copying an (almost) arbitrary list

Suppose now that the list structure to be copied may contain cells pointed to more than once, that is, *sharing* and *cycles* may exist in the list. If Algorithm I were applied to such a structure, the "copy" would be non-isomorphic to the original in that shared cells would be copied as many times as there were paths to them from the head of the list. And if there were cycles in the structure, Algorithm I would loop indefinitely.

The traditional method for dealing with these problems is to plant a "forwarding address" in (say) car of each cell of the original list when it is first visited [1, 4, 7, 9]. The forwarding address points to the copy of the cell in which it is found. If, during tracing of the original structure, a forwarding address should be discovered where an ordinary car was expected, a pointer to that cell could be "forwarded" to the copy, and no spurious copies made. This technique will be used here. Since the new list area is assumed to be a block of contiguous locations, checking for a forwarding address can be accomplished by comparing a pointer with the address boundaries of the region. Let the predicate new(x) be true if and only if x points to a cell in the new list area.

The forwarding address technique has two immediate consequences. First, an old car displaced by a forwarding address must be saved somehow. This is simple when car is an atom, for then old and new cars are the same; but if car is a list, old and new cars will not in general have the same value, so the old value must be salted away in the copy cell. The second consequence is that at some point the forwarding addresses must be removed and the old cars restored. This suggests a second pass over the original structure, whose main purpose is the restoration of old cars.

The second pass must do other work as well. Observe that with each cell of the original list are associated *five* quantities of interest (old and new car, old and new cdr, forwarding address), but only *four* places to put them (car and cdr fields of the original cell and the copy). Obviously, then, not all five items can be stored between passes; some must be computed or recomputed during Pass Two. This computation is greatly simplified by the fact that the second pass can mimic exactly the order in which the first pass visits cells.

It will be convenient to identify pointers as of three types: pointers to atoms, pointers to cells not yet visited during the current pass, and pointers to cells already visited. Let "UV pointer" refer to a pointer to a "UV cell", that is, an unvisited one. An "AV pointer" will point to an "AV cell", that is, an already visited one. These definitions depend, of course, on the *order* in which cells are visited. An AV cell can be identified during the first pass by the presence of a forwarding address in its car. During the second pass, an AV cell is one in which car is *not* a forwarding address, that is, a cell whose forwarding address has already been removed. When multiple pointers to a single cell exist, one of them will be UV, and the rest AV.

We may now make a crucial observation about AV and UV pointers: the new value of any UV pointer can be recomputed during the second pass, but the new value of an AV pointer cannot. Consider first the case of a UV pointer. If it is followed immediately after discovery during the trace, its new value will be a pointer to the following cell in the copy area. If not, it must be car of a cell on the k-list, and its new value will be the value of avail when the cell is removed. In either case the new value can be recalculated during the second pass, provided the two passes visit cells in exactly the same order. Therefore, no new values for UV pointers need to be stored during the first pass.

Now consider an AV pointer. Its new value will be the forwarding address of the AV cell it points to. But during the second pass that forwarding address will be removed *before* the AV pointer is encountered. Thus there is no choice but to store in one of the four available places the new value of each AV pointer found.

If a pointer points to an atom, old and new values are identical, so only one value needs to be saved between passes. If a pointer is a UV pointer, only its old value needs to be saved, since its new value can be recalculated during the second pass. Only in the case of an AV pointer must two values be preserved. Recalling that one of the available storage locations is reserved for a cell's forwarding address, these observations mean that only cells with *two* AV pointers require more than four places to put things. This problem will be dealt with in Section 4; for the moment *assume that there are no such cells.*

Note that if a cell has two UV pointers, only three items need to be saved. The fourth place (the new cdr) can be used, happily, to store a k-list link, just as was done in Algorithm I.

Figure 2 shows the operation of the two-pass algorithm on a list with shared substructures and cycles. Figure 2(a) shows the original structure, and Figure 2(b) shows the state just before the first cell is removed from the k-list. The cars of original cells visited thus far have been replaced by forwarding addresses. Except for cells on the k-list, all AV pointers in the copy have their final values, these having been obtained from the appropriate forwarding addresses. Note that in Figure 2(b) k points to a cell with an AV pointer in its car. When this cell is removed from the k-list, its new car *must* be saved, as was observed above. Since the old car must also be preserved, it is moved to cdr of the new cell, where the k-list link is no longer needed.

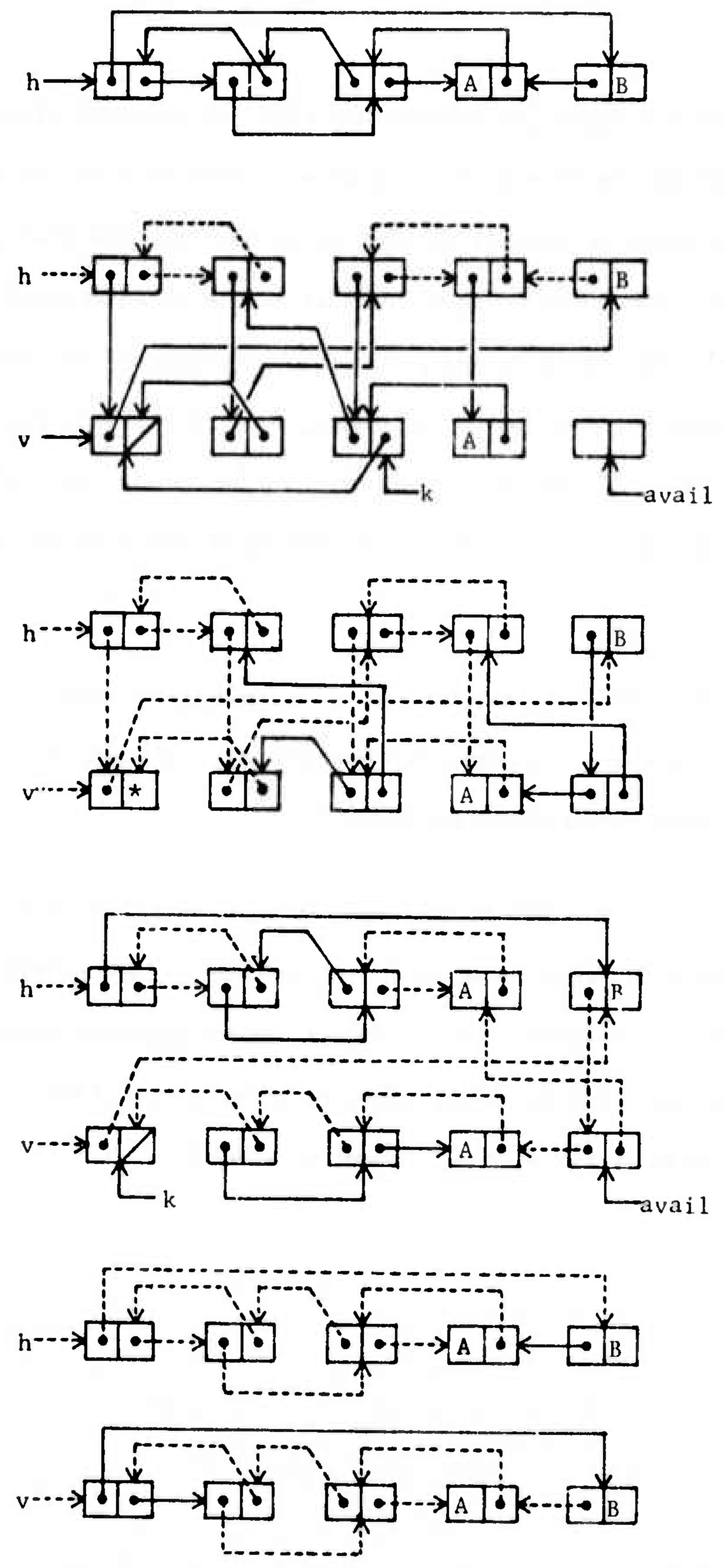


Figure 2: Copying a list with sharing and cycles.

This is shown in Figure 2(c), in which both k-list cells have been removed and Pass One is complete. The "*" in the first new cdr is a marker which indicates that the cell containing it *should* be added to the k-list during Pass Two. This small luxury is made possible by the fact that only three quantities need be stored on behalf of such cells. (A reserved atom is not necessary for this job; any atom will do.) Figure 2(d) shows the progress of Pass Two up to popping the k-list for the first time; it corresponds to Figure 2(b) for Pass One. In Pass Two the k-list contains only those cells *known* to have UV pointers in their cars, in other words, cells found with the atom * in their cdrs.

Algorithm II, below, will copy the list at h into the area beginning at avail, and terminate with v pointing to the copy. Step A5 cannot deal with cells having two AV pointers; this problem will be remedied in Section 4.

A close examination of Step A5 will reveal that during Pass Two, the question of whether an original car or cdr is an AV or a UV pointer can be answered *without* fetching from memory the cell it points to. This permits the eight-way decision made in Step A5 to be repeated in Step C4 by looking only at car and cdr of the original cell and the copy. The importance of this will be seen in Section 4.

ALGORITHM II

Pass One, Part A: Copy the top level of a list.

A1. [Initialize.] k←NIL, v←avail, x←h.

A2. [Get new cell.] n←avail, avail←avail+1.

A3. [Save car and cdr.] a←car(x), d←cdr(x).

A4. [Store forwarding address.] car(x)←n.

A5. [Process cell x.] Find the data types of a and d. If one is a list, check car of the cell it points to to discover whether it is UV or AV. Go to the appropriate box below:

			d	
		atom	AV list	UV list
a	atom	car(n)←a cdr(n)←d go to B1	car(n)←a cdr(n)←car(d) go to B1	car(n)←a cdr(n)←avail x←d
	AV list	car(n)←car(a) cdr(n)←a go to B1		car(n)←a cdr(n)←k k←n x←d
	UV list	car(n)←a cdr(n)←d x←a	car(n)←a cdr(n)←car(d) x←a	

A6. [Loop.] Go to Step A2.

Pass One, Part B: Find the most recently seen sublist.

B1. [Done?] If k=NIL then go to Step C1.

B2. [Pop k-list.] t←k, k←cdr(k), x←car(t).

B3. [x already copied?] If new(car(x)) then cdr(t)←car(t), car(t)←car(x), and go to Step B1.

B4. [Leave marker, return to part A.] cdr(t)←*, and go to Step A2.

Pass Two, Part C: Trace the top level of a list.

C1. [Initialize.] k←NIL, avail←v, x←h.

C2. [Get new cell.] n←avail, avail←avail+1.

C3. [Save car and cdr.] a←car(n), d←cdr(x). (Note car(n), not car(x). Note also that car(n) may not be the original car(x); it might be the original cdr(x) instead. See Steps A5 and B2.)

C4. [Process cell x.] Find the data types of a and d, and go to the appropriate box below:

		d	
		atom	list
a	atom	car(x)←a go to D1	car(x)←a. If cdr(n)=n+1 then x←d Otherwise, go to D1.
	list	If cdr(n) is an atom then car(x)←a, car(n)←avail, x←a. Otherwise, car(x)←cdr(n), cdr(n)←d, and go to D1.	If cdr(n)=* then car(x)←a, cdr(n)←k, k←n, x←d. If new(cdr(n)) then car(x)←a, car(n)←avail, x←a. Otherwise, car(x)←cdr(n), cdr(n)←avail, x←d.

C5. [Loop.] Go to Step C2.

Pass Two, Part D: Find the most recently seen sublist.

D1. [Done?] If k=NIL then halt.

D2. [Pop k-list.] t←k, k←cdr(k), x←car(t), car(t)←avail, cdr(t)←t+1, and go to C2.

4. Copying arbitrary lists

We turn now to the problem of cells with two AV pointers. Since such cells require, according to the current two-pass scheme, that five things be stored in four places, another method must be found for dealing with them. Two passes are not sufficient.

We introduce, therefore, a special pass in between the current two, which will handle these cells. During the first pass, no new values will be stored for them. The copy of such a cell will be in fact an *exact* copy of the original, the old car will get a forwarding address (as it must), and the old cdr will be used to link all such cells together on the "b-list".

After Pass One is finished, the b-list will be processed. When a cell is removed from the b-list, its original car and cdr will be restored by following the forwarding address to the cell's copy. Furthermore, the *new* values of car and cdr are the forwarding addresses of the cells pointed to by the old car and cdr. Final values can therefore be given to both pointers of both cells.

Two questions arise about the use of this technique. First, how can we be sure that the necessary forwarding addresses will still be there? Put another way, is it possible for a b-list cell to contain a pointer to a cell already removed from the b-list, and whose forwarding address, therefore, is lost? It is plainly *not* possible, provided the b-list is processed in LIFO order.

Second, can we still guarantee that the second pass will be an exact replica of the first? The problem here is that during Pass Two certain cells (those previously on the b-list) will appear to have been visited already, when in fact they have not been. But recall from Section 3 that there is enough information in the cell and its copy to reconstruct exactly which cell was visited next during Pass One, and whether pointers pointed to AV or UV cells, *without* looking at the cells pointed to by the original car and cdr. What will happen with former b-list cells is that they will be visited with the expectation of finding a forwarding address in car; if it is missing, then the cell was on the b-list.

Figure 3 shows how the b-list works. Figure 3(a) is the original list, and Figure 3(b) shows the state of affairs just before visiting a cell destined for the b-list. Figure 3(c) shows the situation after the visit, which also marks the end of Pass One. Note that the original car and cdr are stored in the copy of the cell.

Next comes the processing of cells on the b-list, during which final values are written in each original cell and its copy. This is done in Figure 3(d). Pass Two then begins. Figure 3(e) for Pass Two corresponds to Figure 3(b) for Pass One. It also illustrates the hazard of relying on forwarding addresses, instead of the trace order, to set new values of UV pointers. Whereas car(k)←car(car(k)) would have worked in Figure 2(d), it will clearly fail in Figure 3(e) because car(car(k)) is no longer a forwarding address.

Algorithm II can now be extended to cover arbitrary lists. First, cells with two AV pointers should be put on the b-list (b should be initialized to NIL in Step A1). In other words, the empty box in Step A5 should be filled with the following:

```
car(n)←a
cdr(n)←d
cdr(x)←b
b←x
go to B1
```

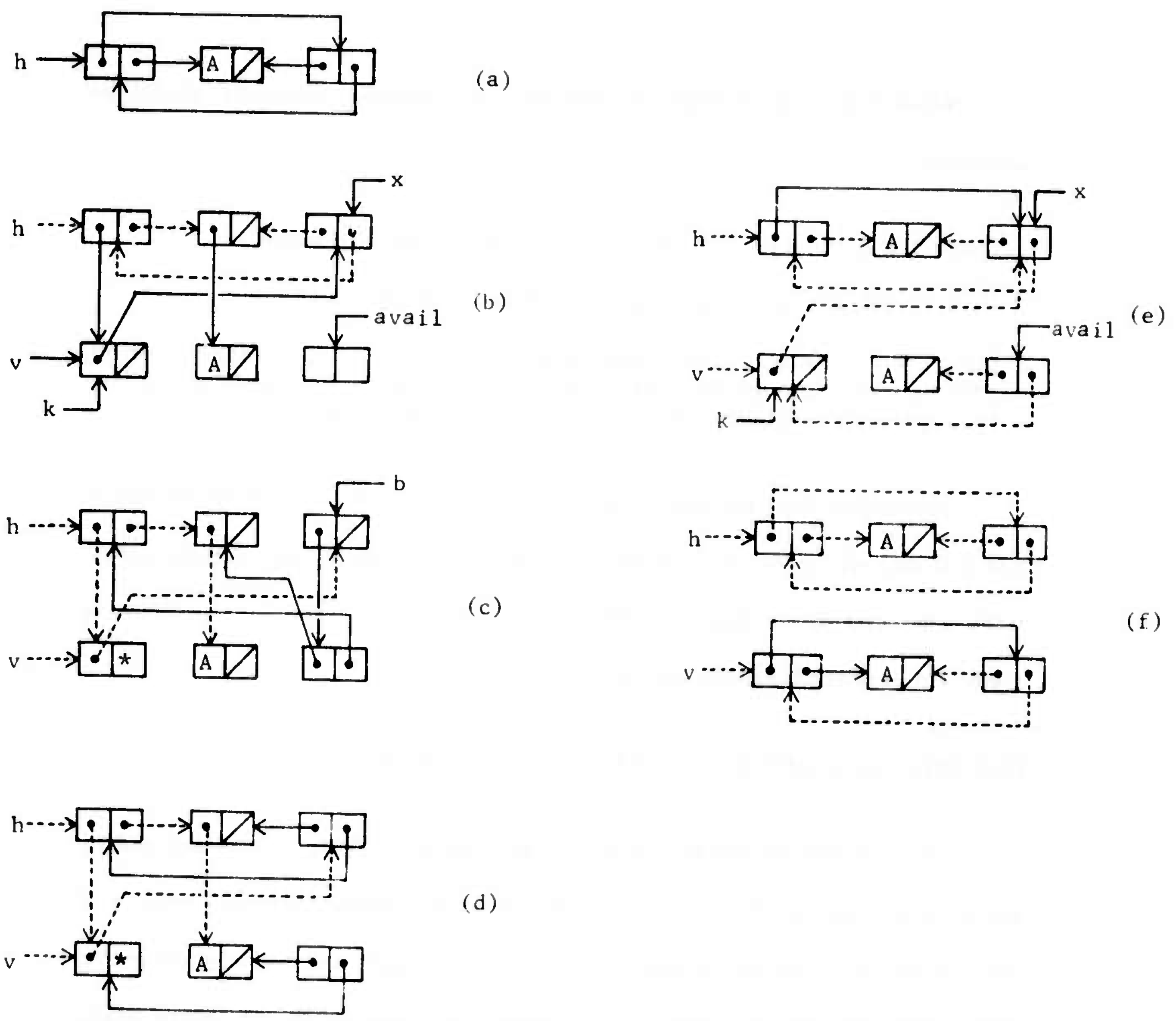


Figure 3: Copying with the b-list.

When Pass One finishes (at Step B1), the following statements should be executed:

1. [Done?] If b=NIL then go to C1. (Begin Pass Two if the b-list is empty.)
2. [Pop b-list.] x←b, b←cdr(b), n←car(x). (n points to the copy of x.)
3. [Set final values and loop.] a←car(n), cdr(x)←cdr(n), car(n)←car(car(n)), cdr(n)←car(cdr(n)), car(x)←a, and go to Step 1. (car(x) is the last to be written because of the possibility that x might contain a pointer to itself.)

One change must be made in Pass Two. Each new value of x must be checked to see if it was on the b-list. If it was, car(x) will not be a forwarding address, and no work will need to be done on either x or n. We insert, therefore, the following statement between Steps C2 and C3:

C2.5 [Was x on b-list?] If new(car(x)) is false then go to Step D1.

Let Algorithm III denote Algorithm II with these additions. The careful reader will have noticed that Pass Two of Algorithm III can be simplified slightly if more cells are put on the b-list during Pass One. Those cells eligible are ones containing one atom pointer and one AV pointer. If AV pointers are rare overall, this change would speed up Pass Two of the algorithm. Addition of these (few) cells to the b-list would free the second pass of having to check for certain cases in Step C4. The savings would thus be proportional to the number of cells with one atom pointer and one list pointer (AV or UV); the cost would be proportional only to the number of cells with one atom pointer and one AV pointer.

5. Comparison with Fisher's algorithm

Because Algorithm III and Fisher's algorithm are both linear in the number of cells copied, a comparison of the two must look at how much computatational work is done per cell, and not just at the number of cell visits. The analysis of this section and the Appendix will measure how many times a list cell must be read or written by each algorithm. This simplifies the comparison task considerably by ignoring such things as arithmetic operations, instruction fetches, and the time required by LISP primitives such as *atom*. For systems in which reading or writing a list cell is expensive relative to, say, fetching an instruction--for example, a system where the copying algorithm is microprogrammed or resides in a cache--the analysis done here realistically measures the computational effort involved. In other systems the work measured here is just part of the total.

The straightforward but tedious counting of memory accesses in the two algorithms is done in the Appendix. The speed of each algorithm depends crucially on the extent of multiple pointers to the same cell. For a given list structure, let a be the fraction of cars which point to lists; let d be the fraction of cdrs which do. Let b be the fraction of cells that go on the b-list in Algorithm III; let k_1 and k_2 be the fractions of cells that go on the k-list during Pass One and Pass Two, respectively. The Appendix shows that Algorithm III will execute

$$T = (5+2a+d+3b+2k_1+2k_2)n$$

reads and writes of list cells to copy an n-cell structure.

Fisher's algorithm (as it appears, slightly optimized, in the Appendix) will execute

$$T' = (10+2a+d+c_1+2c_2)n$$

memory operations on list cells to copy an n-cell structure. The parameter c_1 is the fraction of cells whose cars were UV pointers during the first pass of Fisher's algorithm; c_2 is the fraction of cells whose cdrs were AV pointers. Thus, for structures with little sharing, c_1 is approximately a, and c_2 is approximately zero.

We can easily show that T is always less than T'. Because no cell can be on both the k-list and the b-list in either pass of Algorithm III, $b+k_1 \le 1$ and $b+k_2 \le 1$. Then

$$T = (5+2a+d+2(b+k_1)+(b+k_2)+k_2)n \le (9+2a+d)n < T'.$$

The relationships among the various parameters of T and T' can be seen if we make one simplifying assumption. Observe that Algorithm III and Fisher's algorithm do not visit cells in the same order: where Algorithm III processes deferred list cars in LIFO order (via the k-list), Fisher's algorithm uses FIFO order (via a sequential scan of the copy, in the manner of Cheney [1]). This implies that an AV pointer for one algorithm may turn out to be a UV pointer for the other. The assumption we will make is that the frequency of occurrence of AV pointers in car, in cdr, and in both, will be the same for both algorithms. (Note that the *total* number of AV pointers must be the same for both algorithms, independent of the assumption.)

Under this assumption, Figure 4 illustrates the relationships among the parameters when cells are classified according to the table of Step A5 in Algorithm III. Note that the containment of one area by another (e.g., area b is contained by area c_2) does not mean that *particular* cells in the first category are necessarily in the second; it means only that the *number* of cells in the first is no greater than the number of cells in the second (e.g., $b \le c_2$).

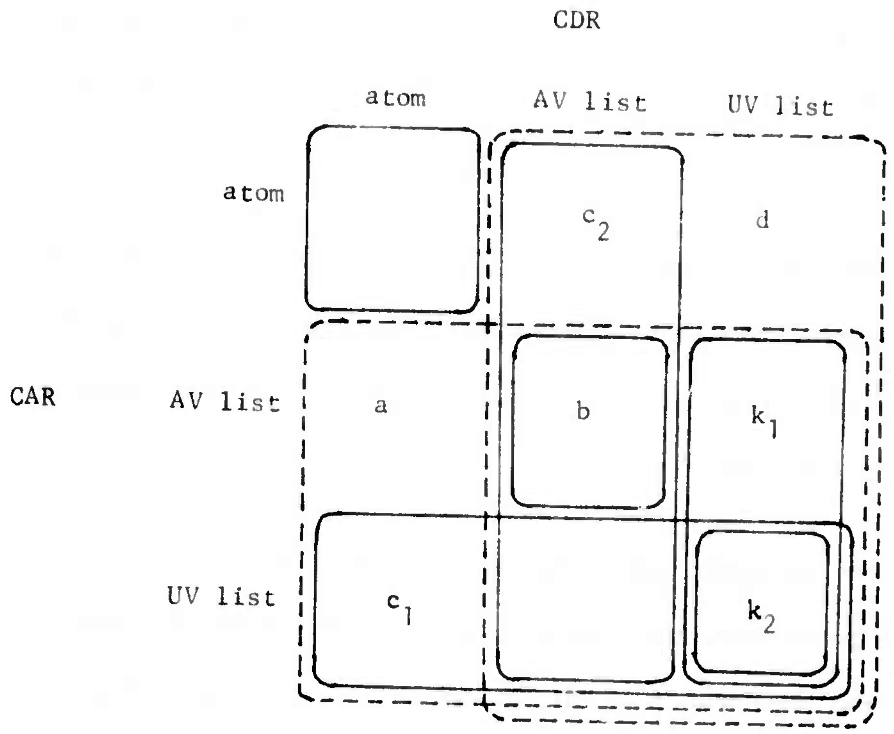


Figure 4: Relationships among the parameters of Algorithm III and Fisher's algorithm.

Table I shows the number of list cell references that each algorithm would make for a variety of n-cell list structures. Both alp ithms are at their best for a linear list of atoms (Table I(a)), but the extremes of relative performance come, as might be guessed, when there is considerable sharing. Table I(c) shows the worst case structure for Algorithm III relative to Fisher's algorithm: a full binary tree whose leaf nodes all contain two AV pointers. For such a structure, Fisher's algorithm would execute 1.26 times as many memory accesses as Algorithm III.

Algorithm III does best in comparison with Fisher's algorithm for the structure of Table I(d): a list in which all cars are UV pointers (except, necessarily, one) and all cdrs are AV pointers. For this structure, twice as many list-cell references take place in Fisher's algorithm as in Algorithm III.

Table I(e) uses parameter values derived from measurements of real list structures in five large LISP programs, reported in [2]. That study found that roughly one third of cars and three fourths of cdrs pointed to lists. This means that the fraction of pointers which were AV was $.333+.75-1=.083$, since for each list cell there is exactly one UV pointer. If we assume that AV pointers were as common among car list pointers as among cdr list pointers, then the fraction of cells containing UV cars is .307, and the fraction containing UV cdrs is .692. This means that $c_1=.307$ and $c_2=.058$. If we assume that car being a list is independent of cdr being a list, then $k_1=.333*.692=.23$ and $k_2=.307*.692=.212$. If AV pointers occur independently in car and cdr, then $b=.026*.058=.002$. For structures with these characteristics, Fisher's algorithm will execute abcut 1.62 times as many list-cell references as will Algorithm III.

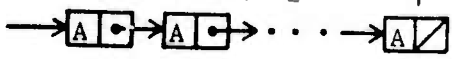
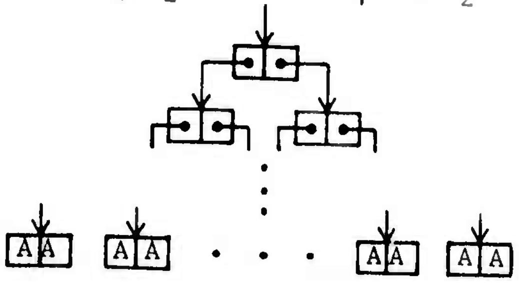
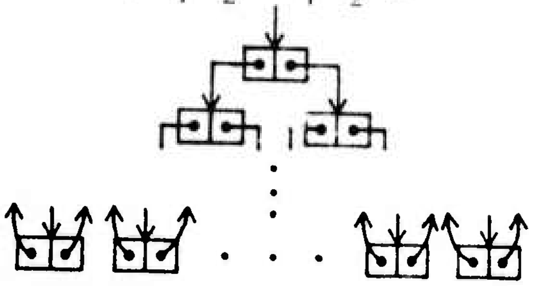
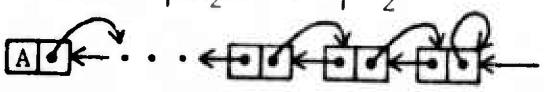
LIST STRUCTURE	Memory References		$\frac{T'}{T}$
	Fisher's algorithm T'	Algorithm III T	
(a) List of atoms $a=0,\ d=1,\ k_1=k_2=b=0,\ c_1=c_2=0$ 	11 n	6 n	1.83
(b) Balanced binary tree $a=d=k_1=k_2=.5,\ b=0,\ c_1=.5,\ c_2=0$ 	12 n	8.5 n	1.41
(c) Worst case for Algorithm III $a=d=1,\ k_1=k_2=b=c_1=c_2=.5$ 	14.5 n	11.5 n	1.26
(d) Worst case for Fisher's algorithm $a=d=1,\ k_1=k_2=b=0,\ c_1=c_2=1$ 	16 n	8 n	2
(e) Typical LISP case from [2] $a=.333,\ d=.75,\ k_1=.23,\ k_2=.212,\ b=.002,\ c_1=.307,\ c_2=.058$	11.84 n	7.3 n	1.62

TABLE I: Comparison of Algorithm III and Fisher's algorithm. Parameter values neglect terms of $O(1/n)$.

6. Conclusion

An algorithm has been given for copying arbitrarily linked LISP-type list structures into a block of sequential memory locations without the use of a stack. Under some simple assumptions about computation time, the algorithm has been shown to be faster than the best previous algorithm, that of Fisher [4]. The speed-up factor is between 1.26 and 2, and for some list structures found in practice is about 1.62.

Acknowledgements. Helpful comments on a draft of this paper were made by Guy Almes, Ellis Cohen, and Samuel Fuller.

APPENDIX: Memory references in Algorithm III and Fisher's algorithm

As discussed in Section 5, we will compare the two algorithms by counting the number of times each must access memory to read or write a list cell. We assume that car and cdr are contained in a single word of memory. We also assume a certain amount of straightforward optimization with respect to this measure, e.g., if car(x) and cdr(x) are both read (written) in a single iteration of an algorithm, we will say that *one* memory read (write) has taken place. Operation counts will be functions of n, the number of cells in the structure to be copied; of a, the fraction of cars which point to lists; of d, the fraction of cdrs which point to lists; and of some additional terms associated with one algorithm or the other.

Analysis of Algorithm III

Pass One. Each cell of the original structure is read once for every pointer to it. The first of these reads occurs when the trace first encounters that cell (Step A3); the rest occur in order to obtain the forwarding address stored there (Steps A5 and B3). The number of list pointers, and therefore the number of reads of original cells, is $an+dn+1$ (the 1 comes from the pointer *root*). More reads will occur when cells are popped off the k-list in Step B2. If k_1 is the fraction of cells which are put on the k-list in Step A5, then another k_1n reads will occur, for a total of $an+dn+k_1n+1$ during Pass One. Write operations occur when each forwarding address is written in Step A4 (n writes), when each new cell is written in Step A5 (another n), and when cells on the k-list have their contents altered in Step B3 or B4 (k_1n writes). Thus $2n+k_1n$ write operations will occur during Pass One.

B-List Pass. Let b be the fraction of cells which are put on the b-list. Then for each b-list cell the following operations will occur: the original cell and its copy will be read and written once each, and the cells pointed to by the original car and cdr will be read to retrieve their forwarding addresses. The totals are 4bn reads and 2bn writes.

Pass Two. Cells formerly on the b-list will be visited once (Step C2.5) and their copies will be neither read nor written. Thus there will be n reads of original cells, but n-bn of copy cells. B-list cells have had their original cars restored already, so n-bn writes will occur for this purpose. Copy cells will be written in Step C4 if car is a list, but since copies of b-list cells never see Step C4, this will contribute an-bn writes. Let k_2 be the fraction of cells which go on the k-list in Step C4 (precisely those with "*" in their cdrs). Then k_2n reads and k_2n writes will occur in Step D2. The totals for Pass Two are $2n-bn+k_2n$ reads and $n+an-2bn+k_2n$ writes

The grand totals for Algorithm III are $2n+an+dn+3bn+k_1n+k_2n+1$ reads and $3n+an+k_1n+k_2n$ writes. Neglecting the single read on behalf of the pointer *root*, the sum is

$$T = (5+2a+d+3b+2k_1+2k_2)n$$

memory accesses.

Fisher's algorithm

Fisher's algorithm, slightly optimized and translated for easier comparison with Algorithm III, appears below. For a thorough discussion of this algorithm, see [4].

Pass One

A1. [Initialize.] v←j←avail, x←root.

A2. [Get free cell.] n←avail, avail←avail+1.

A3. [Process cell x.] car(n)←car(x), d←cdr(x), cdr(x)←n. (The forwarding address is stored in cdr rather than car.) Go to the appropriate box below:

d

atom	AV list	UV list
cdr(n)←d	cdr(n)←cdr(d)	cdr(n)←d, x←d go to A2

A4. [Done?] If j≥avail then go to Step B1.

A5. [Get next car.] x←car(j), j←j+1. (Note the sequential scan of the copy to find sublists.) Go to the appropriate box below:

x

atom	AV list	UV list
go to A4	car(j-1)←cdr(x) go to A4	car(j-1)←avail go to A2

Pass Two

B1. [Initialize.] n←j←v, x←root.

B2. [Exchange cdrs.] d←cdr(n), cdr(x)←d, cdr(n)←x, n←n+1.

B3. [End of segment?] If d is an old list (i.e., not(atom(d)) and not(new(d))), then x←d and go to B2.

B4. [Done?] If j≥n then go to Step C1.

B5. [Check car(j).] If car(j)=n then d←car(cdr(j)), j←j+1, and go to Step B3. Otherwise j←j+1 and go to Step B4. (car(j)=n iff car(j) was the next list traced during Pass One.)

Pass Three

C1. [Scan in reverse order.] n←n-1, x←cdr(n), d←cdr(x).

C2. [Write final cdrs.] d points to an atom, a new list cell (new(d)), or an old list cell. Go to the appropriate box below:

d: atom	d: new list	d: old list
cdr(n)←d	cdr(n)←d cdr(x)←cdr(d)	cdr(n)←n+1

C3. [Done?] If n>v then go to Step C1. Otherwise, avail←j and halt.

Analysis of Fisher's algorithm

Pass One. Just as in Algorithm III, each cell of the original structure is read once for every pointer to it. Thus there are an+dn+1 reads of cells in the original list. In Step A5 each copy cell is read once, so the total number of reads during Pass One is n+an+dn+1. Writes occur when forwarding addresses are written into original cells in Step A3 (n writes), when copy cells are first created in Step A3 (n writes), and when new list cars are written a second time in Step A5 (an writes). The total number of writes is 2n+an.

Pass Two. All new cells are read in Steps B2 and B5, for a total of 2n reads. One original cell is read each time car(j)=n in Step B5. This happens when car(j) was a UV pointer during the Pass One; let c_1 be the fraction of cells for which this occurs. Then $2n+c_1n$ reads occur in Pass Two. Writes happen only in Step B2, in which each cell in the original list and the copy has its cdr written, for a total of 2n.

Pass Three. Each cell in original and copy is read in Step C1, for 2n reads.

Every new cdr is written once in Step C2, for a total of n writes. Whenever d is a new list in Step C2, one additional read and write are done. This condition corresponds to cdr(x) having been an AV pointer during Pass One; let c_2 be the fraction of cells for which this happens. Then there are $2n+c_2n$ reads and $n+c_2n$ writes in Pass Three.

Summing over all three passes, there are $5n+an+dn+c_1n+c_2n+1$ reads and $5n+an+c_2n$ writes. Neglecting the single read associated with *root*, the total is

$$T' = (10+2a+d+c_1+2c_2)n$$

memory accesses.

REFERENCES

1. Cheney, C.J. A Nonrecursive List Compacting Algorithm. *Comm. ACM 13*, 11 (Nov. 1970), 677-678.

2. Clark, D.W. and Green, C.C. An Empirical Study of List Structure in LISP. *Comm. ACM*, to appear.

3. Clark, D.W. A Fast Algorithm for Copying Binary Trees. *Information Processing Letters*, to appear.

4. Fisher, D.A. Copying Cyclic Structures in Linear Time Using Bounded Workspace. *Comm. ACM 18*, 5 (May 1975), 251-252.

5. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1969, pp. 327-328.

6. Lindstrom, G. Scanning List Structures Without Stacks or Tag Bits. *Information Processing Letters 2*, 2 (June 1973), 47-51.

7. Lindstrom, G. Copying List Structures Using Bounded Workspace. *Comm. ACM 17*, 4 (April 1974), 198-202.

8. McCarthy, J. Recursive Functions of Symbolic Expressions and their Computation by Machine-I. *Comm. ACM 3*, 4 (April 1960), 184-195.

9. Minsky, M.L. A LISP Garbage Collector Algorithm Using Serial Secondary Storage. Artificial Intelligence Project Memo 58 (revised), M.I.T. Project MAC, Dec. 1963.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 **REPORT DOCUMENTATION PAGE**

READ INSTRUCTIONS BEFORE COMPLETING FORM

18 1. REPORT NUMBER AFOSR - TR - 76 - 0599	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) 6 COPYING LIST STRUCTURES WITHOUT AUXILIARY STORAGE.	5. TYPE OF REPORT & PERIOD COVERED 9 Interim rept.,
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Douglas W./Clark	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074, w ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Project 1400 Wilson Blvd Arlington, VA 22209	12. REPORT DATE 11 Oct[redacted]975
	13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research Bolling AFB, DC 20032 12 32p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An algorithm is presented for copying an arbitrary list structure into a block of contiguous storage locations without destroying the original list. Apart from a fixed number of program variables, no auxiliary storage, such as a stack, is used. The algorithm needs no mark bits and operates in linear time. It is shown to be significantly faster than the best previous algorithm for the same problem.